WArdian: Securing WebAssembly Applications on Untrusted Mobile Operating Systems

(extended abstract of the MSc dissertation)

José Francisco Oliveira Pinto Malafaia Canana
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor Nuno Miguel Carvalho dos Santos

***Abstract*—Mobile devices are extremely popular and are used for running many data sensitive applications. Thanks to the development of WebAssembly, an emerging browser technology, applications no longer need to be installed on the device. Instead, they can be download on-the-fly and executed directly on the local browser. This is possible because WebAssembly enables web developers to run native C/C++ code on a web page, which runs much faster than typical JavaScript. Using this new approach, the previously implemented sandboxing mechanisms of modern web browsers will not be enough to prevent attacks from an adversary that can compromise the mobile operating system. For instance, if the android OS is attacked by a rootkit, the adversary could compromise the integrity and confidentiality of data manipulated by the sandboxed WebAssembly application. This work presents the design, implementation, and evaluation of *Wardian*, a secure execution environment to run WebAssembly applications on untrusted mobile operating systems. Leveraging on an existent hypervisor, we intend to deconstruct the Chrome browser and run WebAssembly code inside a protected memory space – named *web cage* – where the operating system holds no privileges.**

I. Introduction

Nowadays, mobile devices have a strong influence in our personal connections, and therefore, these devices hold several applications that manage sensitive data, such as banking applications, private social messengers and so on. With the development of WebAssembly [1], an emerging browser technology, developers can now run native C/C++ code directly on the browser and make use of its faster performance when comparing it to typical JavaScript. This technology makes possible the creation of applications that run in the local browser, and therefore do not need to be physically installed on the device. Consequently, browser security is extremely important to prevent adversaries from gaining access to web pages' state and steal users' private information, for instance, prevent access to the user's private keys of a BitCoin wallet software implemented as a downloadable WebAssembly module.

A. Motivation

Currently, Chrome uses a multi-process sandboxing environment to protect the user against attackers that can manipulate web pages and exploit bugs in the renderer process [2]. Due to the nature of web attacks that intend to escape the sandbox environment, Chrome's isolation techniques leverage on the existent operating system (OS) security measures to create and maintain this secured environment. In Linux specifically, the kernel sandboxing environments play an important role in the overall Chrome's isolation and security. In this thesis, we focused on the Chrome browser and how to protect it against attacks on the Android OS.

Unfortunately, modern browsers do not have the necessary security measures implemented to combat an adversary that can compromise the integrity of the operating system of the mobile device [3; 4]. Since the dependency on the operating system security measures is so ingrained in Chrome's sandboxing procedures, an attacker that can successfully compromise the operating system can utilize the browser for more complex attacks on the device. For instance, if the Android OS is attacked by a rootkit, a malicious agent will be able to extract secrets and other sensitive data manipulated by the sandboxed WebAssembly application.

Considering that this class of attacks to the operating system is far from uncommon [5; 6; 7; 8] and that the sophistication of WebAssembly applications will naturally lead to the processing of larger amounts of sensitive data, providing a strong line of defense mechanisms is fundamental.

B. Objectives

Our goal is to address browser security by isolating WebAssembly execution without degrading the overall performance of the Chrome browser nor the underlying operating system's.

Our approach is to combine TrustZone-technology with partitioned execution of the Chrome browser. In particular, by using a TrustZone-assisted hypervisor – *Bao* – and creating isolated memory zones for securing WebAssembly programs execution inside the Chrome browser, we aim to ensure that an untrusted Android OS cannot tamper with the data processed by such programs. As for partitioned execution, the idea is to separate Chrome's renderer process into two parts: one runs normally on the Android OS, while the other partition provides for the instantiation of *web cages*, i.e., trusted execution environments that load and run WebAssembly on a secure memory zone created by *Bao*.

C. Contributions

Our work presents the following contributions:

- A new browser architecture featuring separation of privileges between the Android OS and trusted execution environments for hosting WebAssembly code;
- Implementation of this architecture for the Chrome browser's version targeting mobile devices;
- Improvement on mobile browser security with modest performance degradation;
- An extensive evaluation with real use case web applications.

II. Background

This section provides some necessary background on our work. We begin by introducing WebAssembly and discuss its potential for speeding up the Web. Then, we present the internals of a modern browser – Chrome – focusing in particular on its security mechanisms. Lastly, we present a brief introduction on Bao, our used hypervisor, and its security mechanisms.

A. WebAssembly

WebAssembly, usually shortened to *wasm*, is a low level binary format that can be executed in most of modern browsers. Its small-sized footprint, due to being optimized and precompiled, makes it faster to load and execute than typical JavaScript code (20x-40x faster) [9] which makes it more desirable to use for compute-intensive workloads and all sorts of browser applications [1; 10].

To take advantage of this new technology, web developers can compile their native code into a *wasm module* which will then be glued to the HTML page, using intermediary JavaScript code, so that finally the application can be presented on the browser. In order to compile the C code into *wasm code*, a precompiled toolchain is used, the most popular being Emscripten [11; 12].

Although WebAssembly has many advantages in terms of speed and execution, JavaScript is still necessary to create the connection between the *wasm module* and the browser.

Due to its faster performance, WebAssembly is commonly leveraged to perform compute intensive browser workloads that would otherwise be impossible to accomplish. Since browser and cloud featured applications are on the rise, several native applications are making their transition into the browser [10] to benefit from its mobility and user convenience.

B. Chrome Browser Architecture

Modern browsers, like operating systems, are robust and place each application (renderers, plugins, extensions, etc.) in a separate process that is walled off the concurrent processes. This multi-process architecture ensures that a crash in one component will not disturb the others nor the integrity of the underlying system. It also ensures that each user's access to other user's data is restricted. Essentially, this internal structure brings into the browser the benefits that memory protection and access control have brought to operating systems [13].

To accomplish the desired architecture, Chrome is mainly separated into two different kinds of processes: a privileged *browser process* that is in charge of running the user interface and managing all the other processes, and non-privileged *renderer processes* that are responsible for actually interpreting and laying out the HTML pages. Contrarily to popular belief, the same renderer process can actually allocate several current web pages. By default, one renderer process is created for each live domain instance. To communicate with each other, browser and renderer processes use Chrome's Inter-Process Communication system (IPC) [14], which in Linux and Android is mainly implemented using asynchronous socket pairs.

C. Sandboxing the renderer

To provide for a robust layer of security, and by making use of Chrome's multi-process architecture, each renderer process is enclosed in a separate sandbox environment. This approach is essential to the security of the browser since the renderers process untrusted input and therefore face the risk of being compromised.

To fully isolate the process, the sandboxing environment makes sure to restrict its access to system resources, to the network and to the file system using the host OS built in permissions and a layered approach focused on semantics and attack surface reduction [4]. This layered approach is necessary due to the fact that it is hard to understand semantics when filtering at the system calls interface. There needs to be guarantees that different renderer processes can be distinguished and are unable to affect the integrity of each other while running under different context sandboxes. If this separation was not made, sandbox escapes would be easy to accomplish.

D. Bao Hypervisor

Currently being developed by our colleagues at Universidade do Minho, the *Bao* hypervisor, a lightweight static partitioning tool, is a key component of our proposed architecture, being in charge of managing and creating secure memory zones outside the privileged scope of the Android OS [15]. This lightweight component, leverages on virtual memory and a TrustZone based approach to create and manage enclaves, in our case, guest zones to allocate *web cages* where the main OS holds no privileges. *Bao* also controls the booting process of the Android OS and constantly monitors and manages its behavior, e.g. system calls performed and physical resources used.

The base architecture of *Bao* is purposely simple in order to be easy to integrate in mobile devices and is mostly composed out of two memory zones that run at the *normal world*. The primary zone allocates the Android OS and has the limited ability to instantiate enclaves, i.e other memory zones.

Bao is also in charge of creating and managing a shared memory zone, where both the Android OS and the enclaves

can access, to provide a communication channel for both ends to exchange the necessary information and data needed.

III. Related Work

This section presents the related work, focusing first on existing mechanisms that provide protection against untrusted applications, and then on known mechanisms that can be used for protecting trusted applications from the environment. This second line of work is mostly related to ours.

A. Protection from Untrusted Applications

WebAssembly-based attacks were first introduced in 2017 involving in-browser coin mining [16]. The attackers would take control over the victim's computer by making them visit a malicious web site that would use *wasm code* to leverage on the local machine CPU to mine e-coins. Nowadays, other exploits have surfaced that rely on WebAssembly to compromise the victim's device [17; 18; 19; 20]. Lonkar and Chandrayan [21] performed an in-depth study on several real use cases where WebAssembly was used maliciously, including tech support scams, browser exploits and script-based keyloggers. Next, we describe some relevant approaches proposed in the literature to secure the system against untrusted JavaScript and WebAssembly code executed in the browser.

1) Taint tracking of WebAssembly code: Although security is one of the major requirements for local code execution, sensitive information flow is still at a young development age regarding WebAssembly. To help solve this problem, Szanto, Tamm and Pagnoni [22] developed the first JavaScript based virtual machine for *wasm* execution. The developed JavaScript based virtual machine loads individual *wasm modules* and follows their execution while performing a taint tracking analysis. While *wasm* execution analysis is not one of *WArdian* goals, *wasm* secure execution is. In order to create our isolated *web cages*, the work of Szanto et la. can serve as a valuable foundation to build our desired sandboxing environments.

2) Same Origin Policy and Site Isolation: All modern web browsers, including Chrome, implement *same origin policy* as a security measure to protect websites information from being accessed by different websites. This approach allows scripts in the web page to access or request data from other web pages but only if both share the same origin. With this policy, a malicious script in one web page will not be able to access sensitive data from another web page through it's Document Object Model (DOM).

Oftentimes, skilled adversaries can find vulnerabilities in the code that enforces same origin policy and try to bypass it in order to steal from other websites. *Site isolation*, a recent security feature in Chrome, makes it harder for malicious websites to have access to information outside of their scope [23]. This new approach fully isolates each website, based on its source, in their own renderer process, i.e. in their own sandbox. Without site isolation, a malicious web page with an iframe with login buttons for Facebook would have access to user credentials inside the same renderer, this is, the same process would be managing a vulnerable website and user Facebook credentials. With site isolation, a new renderer process is instantiated for each new website involved in the web page, while showing the same visual information to the user. Currently, Chrome 79 has site isolation [24] enable by default in all desktop environments and isolates all web sites. On Android, only websites that request user credentials are isolated due to performance restrictions of mobile devices.

3) Secure UI: ShadowCrypt [25], a local solution that secures browser textual data, was for many years the state-of-the-art approach for secure I/O within the browser. This solution was implemented as a browser extension and made use of *Shadow DOM* [26] to isolate and encrypt DOM trees. The extension sits between the user and the browser and only provides encrypted private data to the web application. When encrypted information is retrieved from the web page, the data is locally decrypted and shown to the user.

Freyberger et al. [27], explored the limitations of secure I/O systems in the browser, and propose several attacks that successfully bypass *ShadowCrypt*.

Both *ShadowCrypt* and the techniques proposed to evade it are very interesting to study while developing our security approach. Adding the additional untrusted operating system to these case studies makes for a very similar threat model when comparing it to *WArdian*.

4) Browser hardening: As mentioned before, Chrome implements sandboxing mechanisms for confining untrusted web application's code within the boundaries of an unprivileged execution domain. In general, there are several ways to isolate programs and processes from the host operating system, e.g. virtual machines, hardware/software enclaves and so on. Shanmugapriya and Geetha [28] performed an in-depth study on sandboxing environments and how to use them to create lightweight secure environments in large scale systems. This techniques are very useful when running untrustworthy code while making sure it cant have any impact on the underlying operating system and even other sandboxing environments.

However, implementing robust sandboxes is a difficult feat. For instance, in the Chrome browser, several techniques have been attempted to escape the sandbox environments, being the most common ones forged *IPC* messages [23] and memory disclosure attacks [5]. To defend against such threats, most existing approaches involve hardening techniques which consist in the development of customised fixes for the browser [29].

5) Dynamic monitoring of browser extensions: Since extensions execute third party code and have several interactions with user data, several web attacks try to exploit vulnerabilities in these processes, such as private information gathering, browsing history retrieval and password theft. Sanchez-Rola et al. [30] presented two main attacks that attempt to bypass browser security measures related to browser extensions.

Although many possible deviant behaviours by malicious

extensions can be mitigated by the latest isolation and sandboxing environments, user tracking at the network level can still be a problem. M. Weissbacher et al. [31] focused on solving this problem and presented a dynamic technique for identifying user privacy violations on Chrome's extensions based purely on monitoring network traffic patterns.

Chen and Kapravelos [32], also focused on browser extensions related to user privacy, developed a taint analysis framework to perform a large scale study of Chrome's extensions and their privacy practices. Even though [31] used machine learning to identify and analyse network traffic, it remains vulnerable to attackers that possess the ability to mask their network traffic with noise. To solve this issue, the authors present [32], an extension analysis framework that implements dynamic taint tracking for the Chrome browser.

B. Protection from Untrusted Environments

In this section, we switch roles and discuss existing techniques that can protect the execution state of trusted web application code from untrusted environments.

1) Hardware enclaves: A broad class of problems occurs when the adversary can control the operating system, and from there can naturally access the execution state of web applications in the browser. Hardware enclaves can be very successful against compromised browsers and operating systems. Simply put, they provide user-level execution environments for running sensitive code in isolation from the OS. In commodity hardware, hardware enclaves are supported by Intel's software guard extensions (SGX) [33] technology.

Hunt et al. [34] developed a distributed sandbox environment, leveraging hardware enclaves to protect user secret data while it is being processed by untrustworthy services.

When dealing with untrusted clients, web servers cannot rely on the confidentiality and integrity of client-side JavaScript code and data operated on. For example, a local browser JavaScript credit card validation must be made first locally, to warn the user in case of errors, and validated again when it reaches the web server, since it cannot trust the client. This sort of necessary validations adds time to the operations and waste server resources.

TrustJS [35] explores the execution of client-side JavaScript inside a hardware enclave, e.g. Intel SGX, in order to improve user experience and conserve server resources. The developed framework enables trustworthy execution of local JavaScript code that can be attested at the server, instead of validated again.

Fidelius [36], an architecture also based on hardware enclaves, provides user data protection during web browsing sessions. The enclaves are integrated into the browser and enable protection even if the underlying browser and OS are fully controlled by a malicious attacker, a threat model that is very similar to ours. The isolated hardware enclave functions as a small trusted running environment for managing and executing all the web page JavaScript input forms related with user credentials and private actions, e.g. banking transactions.

2) TrustZone-assisted TEEs: Trusted Execution Environments (TEEs) are secure integrity-protected zones that provide processing, memory, and secure storage capabilities for the processes and applications running inside. These environments tend to be isolated from the memory space where the OS runs, which is designated as the *rich execution environment* (REE). Although SGX-based enclaves can be considered an enabling technology for TEE, in mobile computing devices, TEEs are mostly supported by a technology specific to Arm hardware (which is prevalent on mobile platforms). This technology is named ARM TrustZone [37], and it is widely adopted in Android devices as the main isolated zone to run and store sensitive applications and data, e.g. cryptographic keys and certificates [38].

TrustZone relies on *secure* and *normal* worlds, hardware separated, to securely isolate programs and data. Unfortunately, due to the widely adoption of the secure world to run and store applications and data, this secure zone is starting to get bloated and security issues might rise from buggy (or even malicious) code / data stored inside [39]. As an attempt to increase user data security, sandboxing environments that do not rely on the secure world to be isolated from the host OS are worth exploring.

To make TrustZone-assisted TEEs more robust against side channels, Costan et al. [40] introduced a system that provides defenses against known side-channel attacks, such as cache timing attacks and passive address translation attacks by monitoring the memory access patterns of the enclaves while hiding them from the host OS.This sort of implementations is very useful to our own case study, *WArdian*, since we intend to create isolated zones without creating noticeable performance drawbacks to the user.

IV. WARDIAN DESIGN

This section presents *WArdian*, our proposed system for securing WebAssembly code within the Chrome browser against attackers with the ability to control the Android OS. Our solution is targeted to run on Arm platforms featuring ARM TrustZone technology.

A. Motivational Example and Threat Model

Before the technical specifications of our solution, we begin with a demonstrative example showing the need to protect the current state of *wasm* web applications. Consider a bitcoin wallet service that allows its users to manage their account and transactions on a mobile device through the local browser. The local browser, running on the Android OS, has a sandboxed partition where all the WebAssembly code is loaded and executed. When a *wasm module* is requested by the browser, a new sandboxed environment – V8 instance – is created and the application's web server proceeds to send the module requested. In the current state of affairs, an adversary with the ability to control the browser or the OS (e.g., by installing a rootkit), would be able to retrieve sensitive application data, which in this case includes the private key associated with the user's bitcoin wallet.

This problem can be further generalized to other WebAssembly applications that manipulate data items that must be preserved absolutely private to the user. Our work aims to create a separate memory zone to load and execute *wasm modules* while preventing the compromised Android OS from tampering with it. We call these secured WebAssembly packages by the name *cagelets*, and the secure environments where they execute as *web cages*.

1) Threat Model: We want to protect against an attacker that has full control over the operating system of the mobile device. The adversary can also control the browser process, examine and modify unprotected memory where web pages are allocated, and that he/she can examine the communication exchanged between the code executed by the web page and the remote site. We assume, however, that the attacker cannot inspect the contents of secure memory regions allocated to hardware enclaves or to trusted execution environments. It is also worth noticing that the secure memory regions we envision for securing the WebAssembly code are instantiated by the browser process, so we do not consider any type of DOS attacks.

B. System Overview

Our solution tackles the previous problem by isolating the WebAssembly execution from the browser. *WArdian* prevents Chrome from loading and running *wasm modules* and instead leverages on the *Bao* hypervisor, and it's enclave environment, to safely execute *wasm code*.

Bao has the ability to create separate memory spaces inside the same physical device – enclaves – where the Android OS has no privileges, and therefore, cannot access the isolated WebAssembly application's data.

The native Chrome's sandboxing mechanisms remain fully functional, and continue to operate for JavaScript code, but have no indication that WebAssembly is being called by the web pages running inside it's renderer processes. This is accomplished by the introduction of a browser extension that is in charge of tricking web pages into communicating with *WArdian* instead of Chrome's V8 environment.

Running at the enclave, *WArdian* maintains a fully functional WebAssembly runtime that is able to securely communicate with our browser extension, creating the abstraction needed to extract *wasm code*, and inject the corresponding outputs, without V8's knowledge.

Figure 1 presents the architecture of *WArdian*. In the normal world, the Android OS runs as normally expected. The Chrome browser will also have the same behaviour as before, with the exception of renderers that have a *wasm module* loaded. When an IPC message reaches the browser process with a request to download a *wasm module*, a new *web cage* is created inside a new and secure environment where the received *cagelet* is loaded and executed. The underlying mechanisms offered by *Bao* provide memory isolation procedures that prevent an attacker from inspecting the content of a web cage and access the execution state of guest cagelet code.

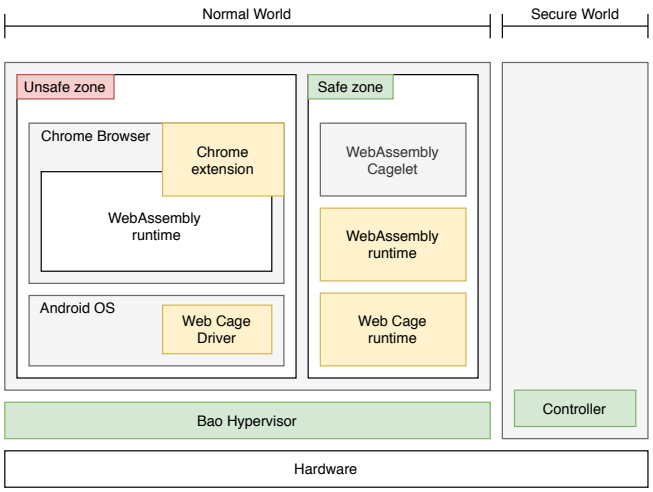

Figure 1. *WArdian* architecture: its specific components are colored in yellow.

Analogous to the first example, consider now a web page application that presents a 2D dice and shows a different dice face each time the user clicks on it, i.e. the faces 1 through 6 are randomized and one is shown on screen. A web developer can take advantage of WebAssembly's near native speed and program the randomizer function in C instead of relying on the browser's JavaScript.

With the *WArdian* extension enabled, the WebAssembly function call is detected and overridden in order to disable Chrome's V8 participation on the request. Instead, the *wasm module* information – *cagelet* – is sent to the web cage environment, via web cage driver and the *Bao* communication mechanisms, and the code is isolated from the browser at the moment of execution. Finally, the *wasm* output is redirected to the *WArdian* extension and consequently injected back into the browser.

With this new WebAssembly life-cycle provided by *WArdian*, *wasm code* is executed outside the privileged bounds of the Android OS which prevents a malicious operating system from tampering with the information managed by the sandboxed *wasm* application, e.g. by tampering with the *wasm module* in order to create a weighted dice (one that always shows the same face).

C. WArdian API

WArdian is not only meant for user local data protection but also to improve the ability of web developers to securely insert *wasm* operations in their web pages.

1) Web Developers: In order for *WArdian* to work correctly, it needs to be able to detect and intercept *wasm* calls made by the current web pages loaded in the browser. To accomplish this task, some HTML and JavaScript syntax restrictions are expected to be followed by web developers. These restrictions demand that the web developer uses a specic WArdian API comprising a few HTML tags and JavaScript function calls. Specically, whenever a web

developer intends to execute *wasm code* on the browser, and isolate it via *WArdian*, he/she must name the parent script function with a default name, and all the necessary arguments, for *WArdian* to correctly detect the call.

The previous dice example can be further explored in order to understand these restrictions. Instead of using typical JavaScript to program the dice randomizer function, WebAssembly allows the developer to use native C code and run it directly in the browser. After compiling it to a *wasm module*, the developer has to fetch the resulting *.wasm* file into the web page and call the desired C function via JavaScript.

The function *WardianFetchAndInstantiate* is the one being detected by the *WArdian* extension and overridden. Without *WArdian* enabled in the browser, this JavaScript function would be interpreted and executed inside Chrome's V8 environment but, with *WArdian* enabled, the entire function's code is overwritten to trigger the enclave environment. The enclave output is then injected into it's return variable.

2) Users: On the other hand, users have no interaction with the system. After successfully installing the *WArdian* extension in Chrome, WebAssembly execution remains completely invisible to the user, providing for a normal and inconspicuous browsing session.

D. Extending the Browser

Our lightweight Chrome extension improves browser *wasm* secure execution without disrupting it's native sandboxing mechanisms. *WArdian* starts by monitoring web requests for new *wasm modules* and overrides the JavaScript functions that fetch and instantiate said modules. After this initial setup, the extension runs idle until it listens for a *wasm* function call. When this condition is triggered, our extension disables the web page ability to run the *wasm* function and sends the respective *wasm module* to the enclave. Finally, the *wasm module* is safely executed in our isolated memory space, the result is returned to the extension and consequently injected into the respective web page.

In order to accomplish these tasks, the extension relies on two main components: a background and a content script. The Background script, as the name implies, runs in the background from the moment Chrome starts and is in charge of listening to various events, such as WebAssembly fetch, instantiation and execution requests, and to establish communication with the enclave, via web cage driver.

To override some of the necessary JavaScript functions, a content script was introduced. Recalling Chrome multi-process architecture, extension processes are analogous to renderer processes in which they run in a separate and isolated process, so the background script does not have the ability to directly communicate or alter the web page content. A content script, however, allows the extension to inject JavaScript code in the current web page which will allow it to run in the same context of the renderer process, making it possible to override native JavaScript functions and establish a communication channel between the web page and the background script.

Looking back at the previous dice example, we can now begin to understand how the extension internals actually work. The content script, when the web page begins to load, and since it runs in the same context as the current renderer, has the ability to parse the web page contents for *wasm* calls and override the JavaScript calls that implement them.

At the same time, the background script is already running and listening for *wasm* events. When the user clicks on the dice, the *wasm* function doesn't run locally since the content script already overridden the necessary JavaScript functions. Instead, the background script sends the *wasm module* over to the web cage environment, at the enclave, where the module is executed and the corresponding output sent back to the background script.

Finally, and after being notified by the background script with the *wasm* response, the content script can inject the new dice value into the return variable of the initial web page's JavaScript function that initiated the entire process.

E. Securing Wasm Execution

With the *WArdian* components at the Android OS explained, we now introduce the enclave environment where the *wasm modules* are loaded and executed isolated from the privileged scope of the device's operating system.

In order to create our desired enclave architecture, two main components are needed, a Web Cage runtime, in charge of managing all system calls at the enclave space and a WebAssembly runtime where *wasm modules* can be loaded and safely executed. Several open source projects were analyzed but since we have a very specific hardware architecture and overall functionality in mind, our choices were very filtered at the starting point already.

Both components needed to support ARM architectures and be as lightweight as possible, in order to minimize the enclave space. Considering our strict requirements and available open source projects, we selected the Zephyr RTOS (real time operating system), by the Linux Foundation, and the WebAssembly Micro Runtime, i.e. WAMR, a Bytecode Alliance project. Zephyr is in charge of managing all the systems calls at the enclave, i.e. it works as a replacement for all the necessary Android OS functions that *WArdian* needs while running at the enclave, and WAMR is a lightweight WebAssembly runtime that interprets and executes *wasm code* in a near native speed while also providing low memory usage.

The combination of Zephyr and WAMR makes for a very small enclave environment but with all the necessary security mechanisms and basic functionality in place that enables *WArdian* to successfully accomplish it's proposed goal.

1) Web Cage Environment: By embedding WAMR in Zephyr, and running this system configuration at our enclave space, we accomplish the desired web cage environment that is able to communicate with our *WArdian* browser extension and safely sandbox *wasm code* executions.

F. Communication Channels

In order for both endpoints to communicate with each other, a shared memory zone is implemented by two underlying components: the web cage driver deployed in the Android's memory space, and the web cage runtime residing in the *Bao* enclave. The former provides an interface to the *WArdian* extension.

In the untrusted Android OS, the web cage driver is necessary to manage all the in and outbound communications between both zones. *Bao* also manages these communications making sure to preserve the integrity of the secure environment. The web cage runtime – Zephyr – provides an interface to the WebAssembly runtime – WAMR – and implements the necessary system functions to receive and reply messages to the web cage driver.

1) Native Messaging: To establish a communication channel between the extension and the enclave, our Web Cage driver needs to be able to exchange messages with the *Wardian* extension in real time. This driver works as a communication bridge between the *WArdian* extension and the enclave environment.

Chrome allows developers to exchange messages between extensions and native applications using their Native Messaging API. Our driver communicates with the extension's background script, via native messaging, and is in charge of receiving the *wasm module* to be executed in the enclave, redirecting this information to the enclave and finally redirecting the enclave output back to the extension.

2) Shared Memory: In regards to the communication between the driver and the web cage environment, *Bao* offers custom services that can be called from the Android OS zone and interpreted at the enclave.

WArdian leverages on these services and introduces two new ones: one to start the enclave environment, i.e. WAMR application execution, and one to send *wasm* data to the running web cage. The first one is called as soon as the Chrome browser starts (event triggered by the *WArdian* extension) and the second one is called each time a *wasm* function is requested by the browser.

To minimize *Bao* space occupation on the mobile device, since they can be very limited in terms of physical memory, the shared memory buffer is initialized with only 8 KB of allocated memory. This small buffer is, most of the times, enough for all the commands and parameters that need to be passed between both ends, but in case of large amounts of data, as can be the case when working with big *wasm modules*, a *chunked* approach is used, i.e. big messages are split in several chunks to be sent one at a time.

G. Secure Bootstrap

To prevent an adversary from disabling *WArdian*'s security mechanisms, it is essential to guarantee the integrity of the entire system's trusted computing base (TCB) upon boot. This TCB includes components in the secure world – the controller – and in the normal world – the *Bao* hypervisor and *WArdian*'s web cage components (see Figure 1). When the mobile device starts its bootstrapping process, the controller firmware is verified (according to a typical trusted boot digital signature validation). Then, after validating the digital signatures of *WArdian*'s software images to be executed in the normal world, the controller switches to the normal world and hands over the control flow to *Bao*'s bootloader. This chain of events ensures that the integrity of *WArdian*'s TCB has not been compromised upon boot.

The remaining of the bootstrapping sequence is as follows. *Bao* starts by allocating space for three different memory regions. One for the Android OS (unsafe zone), one for the *WArdian* enclave (safe zone), and a third memory region where both previous regions have access. Once all three memory zones are allocated, *Bao* starts the Android booting process in the unsafe zone and the enclave environment in the safe zone. The third memory region is intended for the communication process explained in the previous section.

V. Implementation

During the development of *WArdian*, several prototypes were built while trying to accomplish our desired final goal. This section presents all the major steps taken and our reasoning for them to happen. We also present some of the setbacks we faced and how we were able to solve them.

A. WArdian Prototypes

Since *WArdian* has a complex architecture where multiple components are integrated and work together in different ways, we have opted, since the beginning of the development, for a iterative and incremental approach. To follow this development decision, we have built several *WArdian* prototypes and learned from each one how to progress and optimize the next ones.

Two main prototypes were essential for the final *WArdian build*. We have started with QEMU, an open source machine emulator and virtualizer, to simulate our enclave environment and used a generic x86_64 Linux distribution – Ubuntu – as our main operating system (where Chrome runs). This first prototype does not yet run in the desired hardware architecture nor relies on the *Bao* hypervisor but implements all the functionality and communication channels desired.

After the QEMU prototypes, with all the components communication dealt with, we iterated to the second prototyping stage where we started to explore our desired hardware architecture – ARM – and introduced the *Bao* hypervisor in order to create our final *WArdian* architecture.

B. Chrome Modifications

During the projecting phase of this thesis, our main approach was to develop a custom build of the Chrome browser with our desired isolation changes. After some in depth search and testing, we reached to the conclusion that this would not be a good approach for our problem. Due to the nature of the pieces of code that we were trying to modify, some of Chrome's built in security mechanisms were preventing us to accomplish our isolating changes.

However, a Chrome extension, running as a separate Chrome process, has the ability to directly change and communicate with web page's content and consequently bypass the previous problems.

1) Web page content access: In order to detect and intercept WebAssembly calls, we first need to have access to the web page contents, i.e. DOM and JavaScript code. Since Chrome extensions are also sandboxed, and therefore run in individual and separate processes, the *WArdian* extension needs to be granted several browser permissions, as described in its manifest file.

The given permissions enable *WArdian* to have access to all open browser tabs and to be able to distinguish them from each other. This is very important since we need to remain isolating different scripts, from different sources, to have access to each other's content. *WArdian* also has permission to manage web requests to actively monitor and detect *wasm modules* calls.

However, the *WArdian* extension does not yet have the capacity from accessing and actually changing web pages' source code, since separate Chrome processes run in separate contexts. To overcome this issue, and as previously introduced, a content script was necessary. This component enables the extension to run in the same context as the renderer's web page and therefore access all the page's contents

2) Changing JavaScript behavior: With access to the web page's content, we can now modify and insert new code in the current web page. Since we want to override the JavaScript function – *WardianFetchAndInstantiate* – that initiates the entire browser WebAssembly process, we can leverage on our content script to achieve this goal.

Our content script is executed before the web page loads in order to override the desired JavaScript function, that instead of actually fetching and instantiating *wasm* code, now dispatches a *WArdian* custom event that is detected by our background script. When the enclave operations are concluded and the *wasm* result is redirected to the extension, our content script is in charge of injecting it to the web page, via the initial function's return variable.

This new process successfully changes JavaScript behaviour without the knowledge of neither the rendering engine nor the V8 engine, which securely isolates *wasm* code execution from Chrome's scope.

C. Communication with the extension

One unpredictable problem in our QEMU prototypes was the communication channel with the extension, via Web Cage driver. Unfortunately, QEMU does not yet support network communications for their cortex-a53 emulation, this means that we can only communicate via stdin and stdout which severely decreases this set of prototypes performance. Instead of being able to start our QEMU environment and then dynamically send the *wasm modules* and arguments, we have to manually build our WAMR application with the arguments given as header files (.h) and only then start the simulated enclave environment.

Although we have automated this procedure so that our Web Cage driver can start the process and successfully receive the response from the enclave, this process needs to run every time when either the *wasm module* or its arguments change, which is a very frequent event.

However, this problem is only relevant when considering the initial prototypes since our final architecture does not depend on QEMU emulation.

1) Dynamic enclave execution: To fix this issue on the *Bao* prototypes, we built a custom socket connection between the two endpoints: the Web Cage driver and the WAMR application. The Web Cage driver now starts the enclave environment as soon as the extension background process initializes and instead of passing the *wasm module* and arguments via header files, the data is now exchanged via our socket channel which makes the dynamic WAMR execution possible and therefore eliminates the previous performance issue.

D. Bao Integration

Unfortunately, due to time restrictions, we were unable to finish this last prototype in time. This final implementation section presents all the current work in progress aiming at integrating *Bao* and *WArdian* in the same final build.

1) Communication protocols: Out of the box, *Bao* comes with 4 services that the *Bao* application running at the Android OS zone can request: start and close enclave session, invoke enclave command and cancel enclave command. Our *Bao* integration was started by adding two new commands to the custom *Bao* protocol.

As explained before, we need one command to start our WAMR application when a new browser session starts and an additional one (to be called frequently) everytime *wasm* data and its arguments need to be communicated to the web cage environment. To implement these new calls, and with great help from our colleagues at Universidade do Minho, we altered *Bao*'s source code for these changes to take effect.

After instructing our web cage driver to create messages using the custom *Bao* structure it is only a matter of interpreting this information in the web cage environment and execute our WAMR application as we are used to. By adding some new instructions to the source code of *Bao*, specifically on the modules that run at the enclave and interpret receiving messages from the Android OS zone, and by embedding our zephyr final binary in the enclave we could create a similar behaviour known from our previous prototyping experiences.

2) Prototyping problems: Unfortunately, time constraints on the *Bao*'s development team, in combination with project synergies, delayed this final step in the integration of *WArdian* and *Bao*. Due to technical problems while flashing our zephyr binary on the *Bao* enclave, we were unable to have our web cage environment up and running in time to finish our final prototype.

VI. Evaluation

Since the beginning of our work, a big emphasis was placed on device performance due to the major changes made to the WebAssembly browser runtime environment and consequently the *wasm* data life-cycle. In this section we evaluate *WArdian*'s final results based on three main categories.

A. Global Performance

Starting with the overall *WArdian*'s performance, we wanted to measure the global impact of our system in browsing sessions, i.e. the total *wasm* operation's time with and without the *WArdian* extension enabled. To measure this, we changed our developed web pages to record the starting and ending time of the entire *wasm* operation and return the difference via browser console logs.

To perform this evaluation, we used a combination of several developed web pages in order to get an average of *WArdian* execution times. To better understand how our system works, and consequently achieve more interesting conclusions, we started by evaluating our QEMU prototypes and then made our way to the *Bao* ones. Recalling the QEMU performance issues discussed earlier, our gathered benchmark results for this set of prototypes were not surprising.

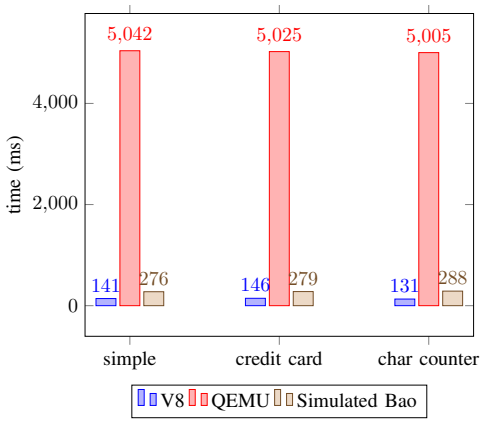

Figure 2. ***WArdian* global performance.**

At this point in time in our *WArdian* evaluation, the *Bao* hypervisor was still in development, which made us have to be creative while gathering the following benchmark values. Predicting the dynamically way in which we can run our WAMR application when using the *Bao* enclave, the need to rebuild our application vanishes, which means that this operation's time can be subtracted from our previous results. To accomplish this new batch of testing, we preemptively added our *wasm* arguments to the WAMR application before the browser *wasm* calls were made. Figure 2 shows the combined results of our batch of tests for the global *WArdian* performance, where we show the average operation's time of several WebAssembly browser calls, while using *WArdian* with QEMU, the simulated *Bao* environment and without the *WArdian* extension enabled, i.e. using the native browser's V8 engine.

Comparing the *WArdian*'s simulated *Bao* prototypes against the native Chrome browser, the results are far more similar and almost unnoticeable to the human eye. Although there is a significant increase in time, approximately 2 times slower, we consider these to be successful *WArdian* results since user experience is unaffected and almost unnoticeable to most users.

B. Micro-benchmarks

Since *WArdian* is a complex system, with several components communicating and working together in different ways, we felt the need to evaluate each component separately, and in specific, measure each communication channel.

The total *WArdian* execution time can be separated into two parts: browser and enclave communication; and WAMR application's execution time. Knowing that the communication channel is composed of several components, we arrived at the following *total time* equation.

$$TotalTime = 2 * (t1 + t2 + t3) + t4 + t5 \quad \textbf{(1)}$$

Our complex communication channel starts at the browser. Chrome passes the *wasm* data to the *WArdian* extension (*t1*), which then needs to be communicated to our web cage driver (*t2*), and only then does it reach the enclave environment (*t3*). The WAMR execution time is represented as *t4* (with *t5* being the application build time) and the communication channel time is doubled since a response with the final *wasm* result needs to be sent back to the browser.

Using this equation, and the gathered results from the previous section, we were able to isolate each communication *bridge* and gather values to evaluate each component's performance.

Web Page	t1	t2	t3	t4	t5	total (ms)
Simple	0.005	0.002	0.354	0.014	4.306	5.042
Card verifier	0.006	0.002	0.437	0.015	4.119	5.025
Counter (S)	0.009	0.001	0.435	0.015	4.103	5.007
Counter (M)	0.010	0.002	0.462	0.015	4.012	4.975
Counter (L)	0.009	0.002	0.451	0.015	4.063	5.002

Table I
WArdian QEMU Micro-benchmarking.

As the results show, and unsurprisingly, the QEMU prototypes present very similar results on all web testing pages. Since QEMU runs in the same space as the underlying operating system, the communication between the web cage driver and the WAMR application is just as simple as two native programs sharing information. We can also detect that the communications between the browser, the *WArdian* extension and the web cage driver (*t1* and *t2*) are very insignificant, which makes sense since these are all Chrome shared processes (the renderer, the extension and the driver),

that communicate via it's inter-process mechanisms[14] and stdin/stdout respectively.

Since the communication between our *WArdian* extension and the web cage driver also happens in the same memory space, it's unsurprisingly fast, and considering that the enclave environment is the same in all *WArdian*'s prototypes, the communication between the driver and the enclave is the most impactful in the total procedure's time.

In regards to the *Bao* values, we once again had to improvise in order to obtain them, since the final prototype was not yet completed at this stage. To accomplish these simulations, we then again removed the compilation time of the WAMR application and used our socket approach to simulate the data transfer between the web cage driver and the enclave environment. We manually ran our testing messages through this channel to simulate the *Bao* shared memory protocol. The communication between the browser and the extension, and the execution time of the *wasm* application, are expected to remain the same between both prototypes.

Web Page	t1	t2	t3	t4	t5	total (ms)
Simple	0.005	0.002	0.124	0.014	NA	0.276
Card verifier	0.006	0.002	0.124	0.015	NA	0.279
Counter (S)	0.009	0.001	0.124	0.015	NA	0.283
Counter (M)	0.010	0.002	0.125	0.015	NA	0.289
Counter (L)	0.009	0.002	0.128	0.015	NA	0.293

Table II
WArdian Simulated Bao Micro-benchmarking.

As our simulated results show, the overall *Bao* total execution time is much shorter, as expected, since the WAMR application can now run dynamically and there is no more need to instantiate a new virtualization environment each time a new *wasm* call is made.

C. Security Analysis

By combining our browser modifications with *Bao*, we successfully mitigated most of the possible attacks potentially issued by a malicious browser or operating system.

Starting with attacks potentially issued by a malicious operating system, *Bao* gives us the memory separation we need to combat and overcome this adversary. By statically partitioning the memory zones of both the Android OS and the enclave environment, *Bao* ensures that the OS cannot access nor override the enclave memory space, which guarantees the safe isolation of all the enclave trusted applications.

Another potentially problematic attack surface is the hardware processor where the android OS and the enclave are running. *Bao* overcomes this issue by allocating, at boot, one single processor core to be assigned to the enclave environment, and the remaining to the Android OS, making it secure from access attempts by a malicious operating system.

Having secured the enclave environment, the *wasm* web applications running at the local browser can still be compromised. *WArdian* deals with this problem by removing all *wasm* processes from the browser's renderer and instead executes *wasm* at the enclave. This ensures that the *wasm* application, and the data it manipulates, are loaded and executed outside the scope of the Android OS, which makes it impossible for it to read or change its contents.

VII. Conclusions

Although *WArdian* is not yet in a stable stage neither for users nor web developers, it already presents a strong foundation for secure WebAssembly browser execution. Even though web developers currently have a well defined set of rules to follow in order to successfully leverage on *WArdian*'s security mechanisms, which might make it overwhelming to implement in their web pages, sensitive *wasm* operations might justify the added work on their part and the consequent lack of performance on the user's device.

Regarding this factor, we should consider that WebAssembly is mostly used to perform intense and complex browser computations, e.g. media decoding, but these are not the kind of operations our system is intended to isolate. *WArdian* was developed to sandbox *wasm* browser operations that leverage and manipulate user sensitive data, like credit card verifications and money transfer operations, which execute faster and therefore will present a least noticeable performance constraint.

The developed Web Cage Driver, in combination with *Bao*, successfully creates a secure communication channel with our enclave environment which makes sure that no sensitive *wasm* information, or data manipulated by the *wasm module*, is leaked to a potentially malicious operating system and/or web browser, preserving the integrity and confidentiality of said data.

Although, in the end, we were not able to successfully finish the integration of *WArdian* and *Bao*, the predicted combination of efforts between the *WArdian* developed components and the used hypervisor present all the initial goals we were trying to accomplish.

In the future, we would like to remove the syntax restrictions and, out of the box, provide support for all the different ways to load and instantiate *wasm* calls making *WArdian* inconspicuous to web developers. Finally, it is worth mentioning that WAMR has native support for Intel's SGX, which means that our final implementation could easily be deployed in desktop environments as well.

Acknowledgments

This work was partially supported by our colleagues and friends at Universidade do Minho, specially David Cerdeira, not only for granting us access to their *Bao* hypervisor implementation but also for the countless hours of help and debugging sessions while trying to integrate their project into ours.

REFERENCES

[1] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

[2] Y. Jia, Z. L. Chua, H. Hu, S. Chen, P. Saxena, and Z. Liang, "The "web/local" boundary is fuzzy: A security study of chrome's process-based sandboxing," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2016.

[3] Google, *Chromium Sandbox Environments*, https://chromium.googlesource.com/chromium/src/+/master/docs/design/sandbox.md.

[4] Chromium, *Chromium Linux Sandboxing*, https://chromium.googlesource.com/chromium/src/+/master/docs/linux/sandboxing.md.

[5] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," *USENIX Security Symposium*, November 2018.

[6] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, 2015.

[7] S. Checkoway and H. Shacham, "Iago attacks: Why the system call api is a bad untrusted rpc interface," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 03 2013.

[8] V. G. Lokhande and D. Vidyarthi, "A study of hardware architecture based attacks to bypass operating system security," in *Security and Privacy*, 2019. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spy2.81

[9] WebAssembly, *FAQ*, 2017, https://webassembly.org/docs/faq/.

[10] ——, *Application examples*, 2019, https://webassembly.eu/.

[11] A. Zakai, "Emscripten: An llvm-to-javascript compiler," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA '11)*, May 2011.

[12] ——, *Emscripten Compiler*, 2016, https://emscripten.org/.

[13] Google, *Chromium Multi-process Architecture*, https://www.chromium.org/developers/design-documents/multi-process-architecture.

[14] ——, *Chromium Inter-process Communication*, https://www.chromium.org/developers/design-documents/inter-process-communication.

[15] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, "Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems," 01 2020.

[16] M. Musch, C. Wressnegger, M. Johns, and K. Rieck, "New kid on the web: A study on the prevalence of webassembly in the wild," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2019.

[17] NVD, *CVE-2018-4121*, 2018, https://nvd.nist.gov/vuln/detail/CVE-2018-4121.

[18] ——, *CVE-2018-4222*, 2018, https://nvd.nist.gov/vuln/detail/CVE-2018-4222.

[19] ——, *CVE-2018-5093*, 2018, https://nvd.nist.gov/vuln/detail/CVE-2018-5093.

[20] ——, *CVE-2018-6092*, 2018, https://nvd.nist.gov/vuln/detail/CVE-2018-6092.

[21] A. Lonkar and S. Chandrayan, *The dark side of WebAssembly*, 2018, https://www.virusbulletin.com/virusbulletin/2018/10/dark-side-webassembly/.

[22] A. Szanto, T. Tamm, and A. Pagnoni, "Taint tracking for webassembly," in *arXiv preprint arXiv:1807.08349*, 07 2018.

[23] C. Reis, A. Moshchuk, and N. Oskov, "Site isolation: Process separation for web sites within the browser," in *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 2019.

[24] Google, *Chromium Site Isolation*, 2019, https://www.chromium.org/Home/chromium-security/site-isolation.

[25] W. He, D. Akhawe, S. Jain, E. Shi, and D. Song, "Shadowcrypt: Encrypted web applications for everyone," *Proceedings of the ACM Conference on Computer and Communications Security*, 11 2014.

[26] Mozzilla, *Shadow DOM*, https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM.

[27] M. Freyberger, W. He, D. Akhawe, M. Mazurek, and P. Mittal, "Cracking shadowcrypt: Exploring the limitations of secure i/o systems in internet browsers," *Proceedings on Privacy Enhancing Technologies*, 04 2018.

[28] K. Shanmugapriya and C. Geetha, "Patterns for it sandbox innovation," in *International Journal of Pure and Applied Mathematics*, 01 2018.

[29] Google, *Meltdown/Spectre*, 2018, https://developers.google.com/web/updates/2018/02/meltdown-spectre.

[30] I. Sanchez-Rola, I. Santos, and D. Balzarotti, "Extension breakdown: Security analysis of browsers extension resources control policies," in *26th USENIX Security Symposium (USENIX Security 17)*, August 2017.

[31] M. Weissbacher, E. Mariconti, G. Suarez-Tangil, G. Stringhini, W. Robertson, and E. Kirda, "Ex-ray: Detection of history-leaking browser extensions," in *Annual Computer Security Applications Conference*, december 2017.

[32] Q. Chen and A. Kapravelos, "Mystique: Uncovering information leakage from browser extensions," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[33] V. Costan and S. Devadas, "Intel sgx explained," 2016. [Online]. Available: https://eprint.iacr.org/2016/086.pdf

[34] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Nov. 2016.

[35] D. Goltzsche, C. Wulf, D. Muthukumaran, K. Rieck, P. Pietzuch, and R. Kapitza, "Trustjs: Trusted client-side execution of javascript," in *Proceedings of the 10th European Workshop on Systems Security*, 2017.

[36] S. Eskandarian, J. Cogan, S. Birnbaum, P. C. W. Brandon, D. Franke, F. Fraser, G. G. Jr., E. Gong, H. T. Nguyen, T. K. Sethi, V. Subbiah, M. Backes, G. Pellegrino, and D. Boneh, "Fidelius: Protecting user secrets from compromised browsers," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[37] ARMSecurityTechnology, *Building a Secure System Using TrustZone Technology*, 2009, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html.

[38] N. Asokan, J.-E. Ekberg, and K. Kostiainen, "The untapped potential of trusted execution environments on mobile devices," in *Financial Cryptography and Data Security*, 2013.

[39] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM Comput. Surv.*

[40] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Security Symposium (USENIX Security 16)*, Aug. 2016.